On Suspending and Resuming Dataflows*

Badrish Chandramouli
Duke University

Christopher N. Bond†
Google Inc.

Shivnath Babu
Duke University

Jun Yang
Duke University

1 Introduction

Consider a long-running, resource-intensive query Q running on a database management system (DBMS). Suppose another task T with much higher priority arrives, and we need to process T as quickly as possible and with all available resources. Ideally, the system should *suspend* the execution of Q, quickly release all resources held by Q, and start T using all resources. Query Q can be *resumed* once T finishes execution, ideally without losing any significant fraction of the work that Q had done prior to suspend.

Beyond supporting mixed-priority DBMS workloads, suspend/resume of queries, or dataflows in general, is important in many other settings:

- *Utility and Grid*: Dataflows now run frequently on computational utilities (e.g., *Condor* [3]) and *Grids* [4] composed of autonomous resources. When the owner of resources wants to use them, dataflows running on these resources must release control quickly, and migrate to other resources.
- *Software rejuvenation*: Benefits of *software rejuvenation* [6], the practice of rebooting enterprise computing systems regularly, are now recognized widely. Reboot is critical when performance degrades due to resource exhaustion caused by resource leaks. Suspend and resume is important in this setting because (1) the challenge of predicting completion times accurately (e.g., see [9]) makes it difficult to schedule task completion to match a rejuvenation schedule; (2) when performance degrades, it may not be cost-effective to wait for all dataflows to complete before rebooting.

Challenges and Contributions Killing and restarting dataflows wastes time and resources, and can lead to starvation if suspend requests are common. For suspend and resume, there are two well-known techniques. First, the dataflow's entire in-memory execution state can be *dumped* to disk on suspend, and read back on resume, like an OS-style process swap. While simple to implement, it can cause high overhead during both suspend and resume, because complex dataflows such as DBMS queries can easily carry gigabytes of in-memory state in modern systems with large memory. Second, the entire state can be *checkpointed* at selected points during execution. Checkpointing minimizes suspend-time overhead, but incurs overhead during execution and resumption. Although the general ideas of dumping and checkpointing have been studied in various systems (e.g., [8]), in this paper we tackle the problem of suspending and resuming complex dataflows consisting of operators with well-understood semantics and behaviors; this additional knowledge introduces unique challenges and opportunities that have not been investigated previously. In particular, we focus on the context of suspending and resuming query execution plans in a DBMS.

For example, consider the simple execution plan for $R \bowtie S \bowtie T$ consisting of two *block-based nested loop joins* (*NLJ*) [5] and three table scans (see Figure 1). We may checkpoint when NLJ_0's heap state (outer buffer) is empty, to reduce checkpointing overhead. However, at this point NLJ_1 may have a full outer buffer. In general, different operators may reach their minimal heap states at different times, making traditional checkpointing very expensive. We argue that operators should perform asynchronous independent checkpointing, but this requires intricate support for precise coordination.

Next, assume a suspend request is received. An operator can choose between dumping its state or *going back* to a previous checkpoint. In Figure 1, NLJ_0 may prefer to write out the few tuples in its outer buffer to disk. But, NLJ_1 may prefer to go back to a checkpoint and reconstruct its state. We argue that different strategies may be appropriate for different operators. The ideal choice depends on the runtime conditions at suspend time.

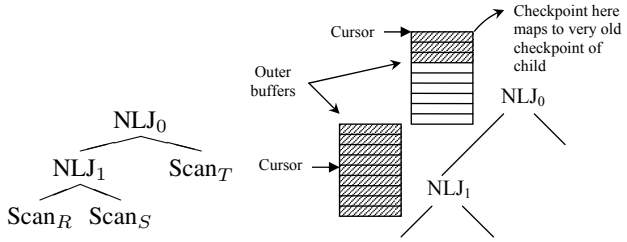

Figure 1. Physical plan & outer buffers.

*This work is supported by an NSF CAREER Award (IIS-0238386) and by IBM Faculty Awards.

†Work performed when the author was at Duke University.

We make the following contributions:

- **Query suspend/resume.** We propose a novel query lifecycle for supporting suspend/resume.
- **Semantics-driven asynchronous checkpointing.** We design strategies for individual plan operators to decide independently when and how to checkpoint in-memory state. We propose a novel low-overhead *contract* mechanism to coordinate independent checkpoints.
- **Online selection of suspend/resume strategies.** We address the problem of choosing the optimal suspend/resume strategy at suspend time to minimize total suspend/resume overhead given a maximum allowed time to suspend.
- **Planning ahead for suspend/resume.** We show how suspend/resume costs and requirements affect different query plans, and motivate the advantage of "suspend-friendly" query planning.
- **Implementation.** We show how our techniques can be incorporated by a DBMS with iterator-based query execution. We have implemented our techniques in *PREDATOR* [11], and experiment results demonstrate their benefits.

2 Overview of Our Approach

To support suspend/resume, we augment the standard *execute* phase of a query in a DBMS with two new phases—*suspend* and *resume*. We address how to support the augmented query lifecycle efficiently.

Compile Phase In this phase, the query optimizer chooses a query plan. Planning could take suspend into account (see Section 3), but our techniques are fully compatible with traditional query optimizers that produce standard plans.

Execute Phase Once the query optimizer chooses a query plan for a query Q, Q enters its execute phase. We augment the execute phase as follows. First, every stateful operator, when it has minimal heap state, performs independent *proactive checkpointing*. For example, NLJ creates a checkpoint each time its outer buffer is empty. This checkpoint has very little associated state, and can be efficiently retained in memory.

Checkpoints can restore an operator O's state, but this is not enough because O still needs its children to resume producing input tuples for O immediately after the point where the checkpoint was created. To address this problem, after creating a checkpoint, O will establish a *contract* with each of its children. A contract is an agreement by the child to (later) produce tuples from the point where the contract was signed. Briefly, the child uses its own previous checkpoint (and some information about its current state) to ensure ability to honor the contract. For instance, when NLJ_1 in Figure 1 receives a contract request from NLJ_0, it maps the contract to its latest proactive checkpoint, and remembers the current outer buffer cursor position.

Stateless operators instead use *reactive checkpointing* [2], done only when the parent requests a contract. We use a graph data structure to keep track of the dependencies among active checkpoints and contracts. The total size of active checkpoints and contracts is small—only $O(n^2)$ for a plan of n operators. We have also developed several optimizations to make suspend/resume more efficient.

Suspend Phase Upon receiving a suspend request, Q enters the *suspend phase*. Each operator in a query plan can choose one of two strategies: *GoBack*, where the operator writes out minimal state and uses a previous checkpoint to resume, and *DumpState*, where the operator dumps its entire state to disk. At suspend time, the DBMS chooses a *suspend plan* for Q, which describes the strategy that will be used to suspend each operator. For example, for the query plan in Figure 1, the suspend plan might choose DumpState for NLJ_0 and GoBack for NLJ_1. The suspend plan and the necessary data to enable resumption are written out to disk as a *SuspendedQuery* structure. After suspend, all of Q's memory resources can be released.

Resume Phase Q enters its *resume phase* when the DBMS is ready to resume it. The goal of this phase is to use the SuspendedQuery structure to reconstruct Q's execution state back to the suspend point, so that the execute phase can continue where it was interrupted. Actions during resume are dictated by the suspend plan used during suspend. For example, if the in-memory state of an NLJ was dumped to disk, we need to read it back into memory. If the NLJ had chosen GoBack to an earlier checkpoint when its outer buffer was empty, then at suspend, we need to ask its outer child to honor the contract and regenerate the content of the outer buffer.

3 Online and Compile-Time Optimization

Every operator needs to decide whether to choose the DumpState or the GoBack strategy. Choosing GoBack often reduces, by orders of magnitude, the state to be written to the disk at suspend time. The downside is a potentially longer resume, because the discarded state needs to be recomputed by the subplan. Given a suspend cost budget, we address the constrained optimization problem that determines the optimal suspend plan with the lowest combined suspend/resume cost. It is ideal to perform this optimization at suspend time, because we have all correct statistics necessary, and we know the exact position of each operator with respect to its contracts and checkpoints. This optimization can be formulated as a mixed-integer program with $O(n^2)$ constraints for a plan with n operators [2]. In practice, we find the optimization time to be negligible.

Some query plans perform better than others in the pres-

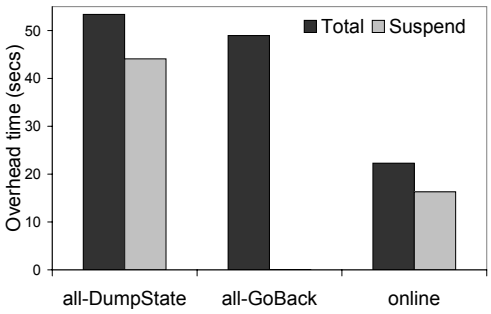

Figure 2. Comparison of approaches.

ence of suspends. We argue that if the plan is chosen without any consideration of suspend/resume, the plan may be suboptimal. If we know the expected suspend pattern, we can choose a query plan tailored for such a situation. For instance, a hybrid hash join (HHJ) usually outperforms a sort-merge join (SMJ) in the absence of suspends. However, HHJ is more costly to suspend and resume because of its in-memory hash table. Thus, SMJ can outperform HHJ in the presence of suspends, and may be preferred if suspends are expected (see [2] for the detailed example).

4 Sample Experiment Results

We implemented our techniques in PREDATOR [11], including checkpointing and contracting, GoBack and DumpState suspend strategies, and the online suspend-plan optimizer. In addition to our online optimization strategy, we experiment with two suspend plans for comparison: (1) all-DumpState, where all operators follow DumpState, and (2) all-GoBack, where all operators perform checkpointing and follow GoBack. In the experiment, we execute, suspend, and resume a complex plan of 10 operators. The suspend plan found by our optimizer is neither of the two extremes (all-GoBack and all-DumpState); it is a hybrid consisting of different strategies for different operators. Figure 2 compares the performance of this plan against all-GoBack and all-DumpState, in terms of total overhead (for both suspend and resume) and suspend cost. We see that the online approach using the hybrid suspend plan performs much better than the purist techniques.

5 Related Work

Our techniques exploit the internal semantics of individual operators to support efficient suspend/resume of complex query plans in their entirety. To the best of our knowledge, we know of no published work that addresses the same problem at such a level. We briefly sample related work here and refer the interested reader to [2].

Researchers have argued [13] for iterating between query optimization and execution. Our work is not intended to handle switching to a different plan after resume. At the same time, plan-switching techniques designed for the query reoptimization setting are generally inadequate for query suspend/resume. Overall, we see our work as complementary and orthogonal to such techniques [1, 10, 12].

Condor *DAGMan* [7] addresses failure recovery for an application consisting of set of tasks with dependencies. While it prevents application restart, it does not support resumption at an intra-task level. Labio et al. [8] considers resuming interrupted data warehouse loads, using logical properties of black-box operators. Our physical approach exposes more optimization opportunities and challenges, and leads to more efficient resume.

6 Conclusion

Many applications benefit from support for suspending and resuming dataflows on demand. This task is challenging especially on modern systems, where dataflows often carry large internal state during execution. We have proposed and implemented a novel lifecycle that supports efficient suspend and resume, taking full advantage of the semantics and behavior of each dataflow operator. In the DBMS setting, we have shown that our techniques add low overhead during normal execution, offer much better suspend/resume performance, and can operate under suspend cost constraints.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, 2000.

[2] B. Chandramouli, C. Bond, S. Babu, and J. Yang. On Suspending and Resuming Queries. Technical report, Duke University, July 2006. `http://www.cs.duke.edu/dbgroup/papers/2006-cbby-qresume.pdf`.

[3] Condor Project. `http://www.cs.wisc.edu/condor/`.

[4] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[5] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[6] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi. Minimizing Completion Time of a Program by Checkpointing and Rejuvenation. In *SIGMETRICS*, 1996.

[7] J. Frey. Condor DAGMan: Handling Inter-Job Dependencies. `http://www.cs.wisc.edu/condor/dagman/`.

[8] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient Resumption of Interrupted Warehouse Loads. In *SIGMOD*, 2000.

[9] G. Luo, J. F. Naughton, C. Ellmann, and M. Watzke. Toward a Progress Indicator for Database Queries. In *SIGMOD*, 2004.

[10] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing Through Progressive Optimization. In *SIGMOD*, 2004.

[11] P. Seshadri. PREDATOR: A Resource for Database Research. In *SIGMOD Record*, 1998.

[12] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2003.

[13] M. Winslett. David DeWitt Speaks Out. In *SIGMOD Record*, 2002.